Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems

Robyn R. Lutz*
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

April 20, 1993

Abstract

This paper analyzes the root causes of safety-related software errors in safety-critical, embedded systems. The results show that software errors identified as potentially hazardous to the system tend to be produced by different error mechanisms than non-safety-related software errors. Safety-related software errors are shown to arise most commonly from (1) discrepancies between the documented requirements specifications and the requirements needed for correct functioning of the system and (2) misunderstandings of the software's interface with the rest of the system. The paper uses these results to identify methods by which requirements errors can be prevented. The goal is to reduce safety-related software errors and to enhance the safety of complex, embedded systems.

I. Introduction

This paper examines 387 software errors uncovered during integration and system testing of two spacecraft, Voyager and Galileo. A software error is defined to be a software-related discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition [6]. Each of these software errors was documented at the time of discovery by a form describing the problem or failure. The form also recorded the subsequent analysis and the corrective actions taken.

As part of the standard procedure for correcting each reported software error, the failure effect of each is classified as negligible, significant, or catastrophic. Those classified as significant or catastrophic are investigated by a systems safety analyst as representing potential safety hazards [13]. For this study the 74 software errors on Voyager and 121 software errors

* Author's mailing address is Dept. of Computer Science, Iowa State University, Ames, IA 50011. The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with NASA.

on Galileo documented as having potentially significant or catastrophic effects arc classified as safety-mlatcd.

The spacecrafts' software is safety- critical in that it monitors and controls components that can be involved in hazardous system behavior [1 I]. The software must execute in a system context without contributing unacceptable risk.

Each spacecraft involves embedded software distributed on several different flight computers. Voyager has roughly 18,000 lines of source code; Galileo has over 22,000 [1 8]. Embedded software is software that runs 011 a computer system that is integral to a larger system whose primary purpose is not computational [G]. The software on both spacecraft is highly interactive in terms of the degree of message-passing among system components, the need to respond in real-tilnc to monitoring of the hardware and environment, and the complex timing issues among parts of the system. The software development for each spacecraft involved multiple teams working for a period of years.

The purpose of this paper is to identify the extent and ways in which the cause/cflcct relationships of safety-rclatccl software errors differ from the cause/cfTcct relationships of lloI]-safety-related software errors. Preliminary results were reported in [1 4]. In particular, the analysis shows that errors in identifying or understanding functional and interface requirements frequently lead to safety-rc]atcd software errors. This distinction is used to identify methods by which the common causes of safety-rc]atcd software errors can be targeted during development. The goal is to improve system safety by understanding and, where possible, removing the prevalent sources of safety-related software errors.

II. Methodology

The study described here characterizes the root causes of the safety-rclatccl software errors discovered during integration and system testing. The recent work by Nakajo and Kume on software error callsc/effect relationships Offers an appropriate framework for classifying the software errors [] 6]. Their work is extended here to account for the additional complexities operative in large, safety-critical, embedded systems with evolving requirements driven by hardware and environmental issues.

As will be seen in Section IV, previous studies of the causes of software errors have dealt primarily with fairly simple, non-embedded systems in familiar application domains. Requirements specifications in these studies generally have been assumed to be correct, and safety issues have not been distinguished from program correctness. The work presented here instead builds 011 that in [1 G] to analyze software errors in safety-critical, embedded systems with developing requirements.

Nakajo and Kume's classification scheme analyzes three points in the path from a software error backwards to its sources. This approach allows classification not only of the documented software error (called the program fault), but also of the earlier human error (the root cause, e.g., a misunderstanding of an interface specification), and, before that, of the process flaws that contribute to the likelihood of the error's occurrence (e.g., inadequate communication between systems engineering and software development teams).

The classification scheme thus leads backwards in time from the evident software error to an analysis of the root cause (usually a communication error or an error in recognizing or deploying requirements), to an analysis of the software development process. By comparing common error mechanisms for the software errors identified as potentially hazardous with those of the other software errors, the prevalent root causes of the safety-related errors are isolated. The classification of the sources of error then is applied here to determine countermeasures which may prevent similar error occurrences in other safety-critical, embedded systems. This paper thus uses the classification scheme to assemble an error profile of safety-related software errors and to identify development methods by which these sources of error can be controlled.

II. Analysis of Safety-Related Software Errors

A. Overview of Classification Scheme

An overview of the classification scheme follows, adjusted to the needs of safety-critical, embedded software. See [16] for additional details on how errors are categorized. An ongoing, multi-project investigation will address the issue of repeatability (do different analysts classify a given error in the same way?).

- Program Faults (Documented Software Errors)

  A. Internal Faults (e.g., syntax)

  B. Interface Faults (interactions with other' system components, such as transfer of data or control)

  C. Functional Faults (operating faults: omission or unnecessary operations; conditional faults: incorrect condition or limit values; behavioral faults: incorrect behavior, not conforming to requirements)

- Human Errors (Root Causes)

  A. Coding or Editing Errors

  B1. Communication Errors Within a Team (misunderstanding software interface specifications)

  B2. Communication Errors Between Teams (misunderstanding hardware interface specifications or other team's software specifications)

  C1. Errors in Recognizing Requirements (misunderstanding specifications or problem domain)

  C2. Errors in Deploying Requirements (problems implementing or translating requirements into a design)

- Process Flaws (Flaws in Control of System Complexity + Inadequacies in Communication or Development Methods)

  A. Inadequate Code Inspection and Testing Methods

B1. inadequate Interface Specifications + Inadequate Communication (among software developers)

B2. Inadequate Interface Specifications + Inadequate Communication (between software and hardware developers)

C1. Requirements Not Identified or Understood + Incomplete Documentation

C2. Requirements Not Identified or Understood + Inadequate Design

Clearly, the attribution of a key human error and a key process flaw to each software error oversimplifies the cause/effect relationship. However, the identification of these factors allows the characterization of safety-related software errors in a way that relates features of the development process and of the system under development to the safety consequences of those features. Similarly the association of each software error with a human error, while unrealistic (in what sense is a failure to predict details of system behavior an error?), allows a useful association between human factors (such as misunderstanding the requirements or the underlying physical realities) and their safety-related consequences.

B. Program Faults

Table 1 (see Appendix) shows the proportion and number (in parentheses) of software errors in the three main categories of Internal Faults, Interface Faults, and Functional Faults for both spacecraft. The right-hand columns provide the same information for only the safety-related software errors.

Safety-related software errors account for 55% of the total software errors for Voyager and 48% of the total software errors for Galileo discovered during integration and system testing.

Few *internal* faults (e.g., coding errors internal to a software module) were uncovered during integration and system testing. An examination of software errors found later during operations also shows few internal errors. It appears that these coding errors are being detected and corrected before system testing begins. They thus are not discussed further in this paper.

At the high level of detail in Table 1, safety-related and non-safety-related software errors display similar proportions of interface and functional faults. *Functional faults* (operating, conditional, or behavioral discrepancies with the functional requirements) are the most common kind of software error.

Table 2 examines the predominant type of program fault, the functional fault, in more detail. At the level of detail in Table 2, differences between the spacecraft begin to appear. On Voyager fully half the safety-related functional faults are attributable to behavioral faults (the software behaving incorrectly). On Galileo, a slightly greater percentage is due to operating faults (nearly always a required but omitted operation in the software) than to behavioral faults. often the omitted operation involves the failure to perform adequate reasonableness checks on data, input to a module. This frequently results in an error-recovery routine being called inappropriately.

Table 2 shows that conditional faults (nearly always erroneous values on conditions or limits) tend to be safety-related on both spacecraft (73% total). Even though adjusting the values of limit variables during testing is considered to be fairly routine, the case of change obscures the difficulty of determining the appropriate value and the safety-related consequences of an inappropriate limit value.

Erroneous values (e.g., of deadbands or delay timers) often involve risk to the spacecraft by causing inappropriate trigger-ing of an error-rccovcry response or by failing to trigger a needed response. The association between conditional faults and safety-rclatccl software errors emphasizes the importance of specifying the correct values for any data used in control decisions in safety-critical, embedded software,

The analysis summarized in 'J'able 1 also identifies *interface faults* (incorrect, interactions with other system components, such as the timing or transfer of data or control) as a significant problem (35% of the safety-related program faults on Voyager; 1970 on Galileo). Sect. IV below describes how the high incidence of interface faults in these complex, embedded systems contrasts with the low incidence of interface faults in earlier studies on simpler, standalone software.

C. Relationships Between Program Faults and Root Causes

The second step in the cause/cfl"cct analysis is to trace backwards in time to the human factors involved in the program faults that were discovered during integration and system testing. Tables 3a and 3b summarize the relationships between the major types of program faults and the key contributing causes.

For *interface faults*, the major human factors are either communication errors within a development team or communication errors between a development team and other teams. In the latter case, a further distinction is made between misunderstanding hardware interface specifications and misunderstanding the interface specifications of other software components. From '1'able 3a it can be seen that communication errors between development teams (rather than within teams) is the leading cause of interface faults (93% on Voyager, 72% on Galileo). Safety-related interface faults arc associated overwhelmingly with *communication errors* between a development team and others (often between software developers and systems engineers), rather than with communication errors within a team.

Significant differences appear in the distribution of fault causes between safety-related and)lon-safety-related inter-face faults. The primary cause of safety-rclatccl *interface* faults is *misunderstood hardware interface specifications* (65% on Voyager; 48% on Galileo). Examples arc faults caused by wrong assumptions about the initial state of relays or by unexpected heartbeat timing patterns in a particular operating mode. On the other hand, the root causes of Ilon-safety-related interface faults arc distributed more evenly between misunderstood hardware specifications and misunderstood software specifications. The profile of safety - related interface faults assembled in Table 3a emphasizes the importance of understanding the software as a set of embedded components in a larger system.

The primary cause of safety-related *functional* faults is errors in *recognizing (understanding) the requirements* (62% on Voyager, 79% on Galileo). On the other hand, non-safety - related functional faults arc more often caused by errors in deploying- implementing- -the requirements.

More specifically, safety-related *conditional* faults (erroneous condition or limit values) arc almost always caused by errors in *recognizing requirements.* Safety-rclatcc] *operational* faults (usually the omission of a required operation) and *behavioral* faults are also caused by errors in recognizing requirements more often than by errors in deploying requirements. Table 3b reflects deficiencies in the documented requirements as well as instances of unknown (at the time of requirements specification) but necessary requirements for the two spacecraft. Section V examines how and to what extent the discovery of requirements during testing can be avoided in safety-critical, embedded systems.

In summary, difficulties with requirements is the key root cause of the safety-related software errors which have persisted until integration and system testing. The tables point to errors in understanding the requirements specifications for the software/system interfaces as the major cause of safety-related interface faults. Similarly, errors in recognizing the requirements is the major root cause leading to safety-related functional faults.

D. Relationships Between Root Causes and Process Flaws

In tracing backwards from the program faults to their sources, features of the system-development process can be identified which facilitate or enable the occurrence of errors. Discrepancies between the difficulty of the problem and the means used to solve it may permit hazardous software errors to occur [4].

The third step of the error analysis therefore associates a pair of process flaws with each program fault [16]. The first element in the pair identifies a process flaw or inadequacy in the *control of the system complexity* (e.g., requirements which are not discovered until system testing). The second element of the pair identifies an associated process flaw in the *communication or development methods used* (e.g., imprecise or unsystematic specification methods).

The two elements of the process-flaw pair are closely related. Frequently, as will be seen in Sect. V, a solution to one flaw will provide a solution to the related flaw. For example, the lack of standardization evidenced by an ambiguous interface specification (an inadequacy in the control of system complexity) and the gap in interteam communication evidenced by a misunderstood interface specification (an inadequacy in the communication methods used) might both be addressed by the project-wide adoption of the same CASE tool.

Table 4a summarizes the relationships between process flaws and the most common causes of interface faults (misunderstanding software interface specifications and misunderstanding hardware interface specifications). The rightmost columns provide information about the safety-related interface faults.

For safety-related *interface faults*, the most common complexity-control flaw is *interfaces not adequately identified or understood* (54% on Voyager; 87% on Galileo). The most common safety-related flaw in the communication or development methods used on Voyager is *hardware behavior not documented* (46%). On Galileo the most common safety-related flaws are *lack of communication between hardware and software teams* (35%) and *interface specifications known but not documented or communicated* (35%).

Anomalous hardware behavior is a more significant factor in safety-related than in non-safety-related interface faults. It is often associated with interface design during system testing, another indication of a unstable software product.

There are significant, variations in the process flaws that cause errors between the two spacecraft. Interface design during testing is involved in almost one-fifth of the safety-critical interface faults on Voyager, but in none of them on Galileo. This is because on Voyager a set of related hardware problems generated nearly half the safety-related interface faults. On the other hand, the problem of interface specifications that are known but not documented is more common on Galileo. This is perhaps due to the increased complexity of the Galileo interfaces.

Table 4b summarizes the relationships between process flaws and the major root causes of functional faults (recognizing and deploying requirements).

For *functional faults,* requirements not identified and requirements not understood are the most common complexity-control flaws. Safety-related functional faults are more likely than non-safety-related functional faults to be caused by *requirements which have not been identified.*

With regard to flaws in the communication or development methods, *missing requirements* are involved in nearly half (42%) the safety-related errors that involve recognizing requirements. *Inadequate design* is the most common flaw leading to errors in deploying requirements on Voyager. On Galileo, *incomplete documentation of requirements* is as important a factor for safety- related errors, but not for non-safety-related errors.

Imprecise or unsystematic specifications are twice as likely to be associated with safety - related functional faults as with non-safety-related functional faults. Similarly, *unknown, undocumented, or wrong requirements* are a greater cause of safety-related than of non-safety-related errors.

These results suggest that the sources of safety-related software errors lie farther back in the software development process- in inadequate requirements- whereas the sources of non-safety-related errors more commonly involve inadequacies in the design phase.

IV. Comparison of Results with Previous Work

Although software errors and their causes have been studied extensively, the current work differs from most of the prior investigations in the four following ways:

1) The software chosen to analyze in most studies is not embedded in a complex system as it is here. The consequence is that the role of interface specifications in controlling software hazards has been underestimated.

2) Unlike the current paper, most studies have analyzed fairly simple systems in familiar and well-understood application domains. Consequently, few software errors have occurred during system testing in most studies, leading to a gap in knowledge regarding the sources of these more-persistent and often more hazardous errors.

3) Most studies assume that the requirements specification is correct. On the spacecraft, as in many large, complex systems, the requirements evolve as knowledge of the system's behavior and the problem domain evolve, Similarly, most studies assume that requirements are fixed by the time that systems testing begins. This leads to a underestimation of the impact of unknown requirements on the scope and schedule of the later stages of the software development process.

4) The distinction between causes of safety-critical and non-safety-critical software errors has not been adequately investigated. Efforts to enhance system safety by specifically targeting the causes of safety-related errors, as distinguished from the causes of all errors, can take

advantage of the distinct error mechanisms, as described in Sect. 5.

A brief description of the scope and results of some related work is given below and compared with the results presented in this paper for safet y-critical, embedded computer systems.

Nakajo anti Kume categorized 670 errors found during the software development of two firmware products for controlling measuring instruments and two software products for instrument measurement programs [16]. over 90% of the errors were either interface or functional faults, similar to the results reported here.

Unlike the results described here, Nakajo and Kume found many conditional faults. It may be that unit testing, as on the spat.ccraft, finds many of the conditional faults prior to system testing. While the key human error on the spacecraft involved communication between teams, the key human error in their study involved communication within a development team. Both studies identified complexity and documentation deficiencies as issues. However, the software errors on the spacecraft tended to involve inherent technical complexity, while the errors identified in the earlier study involved complex correspondences between requirements and their implement ation.

Finally, the key process flaw that they identified was a lack of methods to record known interfaces and describe known functions. in the safety-critical, embedded software on the spacecraft, the flaw was more often a failure to identify or to understand the requirements.

Ostrand anti Weyuker categorized 173 errors found during the development and testing of an editor system [19]. Only 2% of the errors were found during system testing, reflecting the simplicity and stability of the interfaces and requirements. Most of the errors (G]%) were found instead during function testing, Over half these errors were caused by omissions, confirming the findings of the present study that omissions are a major cause of software errors.

Schneidewind and Hoffmann [21] categorized 173 errors found during the development of four small programs by a single programmer. Again, there were no significant interfaces with hardware and little system testing. The most frequent class of errors, other than coding and clerical, was design errors. All three of the most common design errors- extreme conditions neglected, forgotten cases or steps, and loop control errors- are also common functional faults on the spacecraft.

Both the findings presented in [19, 21] ant] in this paper confirm the common experience that early insertion and late discovery of software errors maximizes the time and effort that the correction takes. Errors inserted in the requirements and design phases take longer to find and correct than those inserted in later phases (because they tend to involve complex software structures). Errors discovered in the testing phase take longer to correct (because they tend to be more complicated and difficult to isolate). This is consistent with the results in [1 8] indicating that more severe errors take longer to discover than less severe errors during systcIn-level testing. Furthermore, this effect was found to be more pronounced in more complex (as measured by lines of code) software.

The work done by Endres is a direct forerunner of Nakajo and Kume's in that Endres backtracked from the error type to the technical and organizational causes which led to each type of error [4]. Moreover, because he studied the system testing of an operating system, the software's interaction with the hardware was a source of concern. Endres noted the difficulty of precisely specifying functional demands on the systems before the programmer had seen their effect on the dynamic behavior of the system. His conclusion that better tools were needed to attack this problem still holds true eighteen years after he published his study.

Of the 432 errors that Endres analyzed, 46% were errors in understanding or communicating the problem, or in the choice of a solution, 38% were errors in implementing a solution, and the remaining 16% were coding errors. These results are consistent with the finding here that software with many system interfaces displays a higher percentage of software errors involving understanding requirements or the system implications of alternative solutions.

Eckhardt et al., in a study of software redundancy, analyzed the errors in twenty independent versions of a software component of an inertial navigation system [3]. He found that inadequate understanding of the specifications or the underlying coordinate system was a major contributor to the program faults causing coincident failures.

Addy, looking at the types of errors that caused safety problems in a large, real-time control system, concluded that the design complexity inherent in such a system requires hidden interfaces which allow errors in non-critical software to affect safety-critical software [1]. This is consistent with Selby and Basili's results when they analyzed 770 software errors during the updating of a library tool [22]. Of the 46 errors documented in trouble reports, 70% were categorized as "wrong" and 28% as "missing." They found that subsystems that were highly interactive with other subsystems had proportionately more errors than less interactive subsystems.

Leveson listed a set of common assumptions that are often false for control systems, resulting in software errors [11]. Among these assumptions are that the software specification is correct, that it is possible to predict realistically the software's execution environment (e.g., the existence of transients), and that it is possible to anticipate and specify correctly the software's behavior under all possible circumstances. These assumptions tend to be true for the simple systems in which software errors have been analyzed to date and false for spacecraft and other large, safety-critical, embedded systems. Thus, while studies of software errors in simple systems can assist in understanding internal errors or some functional errors, they are of less help in understanding the causes of safety-related software errors, which tend heavily to involve interfaces or recognition of complex requirements.

Similarly, standard measures of the internal complexity of modules have limited usefulness in anticipating software errors during system testing. It is not the internal complexity of a module but the complexity of the module's connection to its environment that yields the persistent, safety-related errors seen in the embedded systems here [8].

V. Conclusion

A. Recommendations

The results in Sect. III indicate that safety-related software errors tend to be produced by different error mechanisms than non-safety-related software errors. This means that system safety can be directly enhanced by targeting the causes of safety-related errors. Specifically, the following six recommendations emerge from our analysis of safety-related errors in complex, embedded systems.

1. Focus on the interfaces between the software and the system in analyzing the problem domain, since these interfaces are a major source of safety-related software errors.

The traditional goal of the requirements analysis phase is the specification of the software's external interface to the user. This definition is inadequate when the software is deeply embedded in larger systems such as spacecraft, advanced aircraft, air-traffic control units, or manufacturing process-control facilities. in such systems, the software is often physically and logically distributed among various hardware components of the system. The hardware involved may be not only computers but also sensors, actuators, gyros, and science instruments [9].

Specifying the external behavior of the software (its transformation of software inputs into software outputs) only makes sense if the interfaces between the system inputs (e.g., environmental conditions, power transients) and the software inputs (e.g., monitor data) are also specified. Similarly, specifying the interfaces- especially the timing and dependency relationships- between the software outputs (e. g., star identification) and system outputs (e.g., closing the shutter on the star scanner) is necessary. [5,10]

System-development issues such as timing (real-time activities, interrupt handling, frequency of sensor data.), hardware capabilities and limitations (storage capacity, power transients, noise characteristics), communication links (buffer and interface formats), and the expected operating environment (temperature, pressure, radiation) need to be reflected in the software requirements specifications because they are frequently sources of safety-critical software interface errors.

Timing is a particularly difficult source of safety-related software interface faults since timing issues are so often integral to the functional correctness of safety-critical, embedded systems. Timing dependencies (e.g., how long input data is valid for making control decisions) should be included in the software interface specifications. Analytical models or simulations to understand system interfaces are particularly useful for complex, embedded systems.

2. Identify sa~ety-critical hazards early in the requirements analysis.

These hazards are constraints on the possible designs and factors in any contemplated tradeoffs between safety (which tends to encourage software simplicity) and increased functionality (which tends to encourage software complexity) [10, 22]. Many of the safety-related software errors reported in Sect. 111 involve data objects or processes that would be targeted for special attention using hazard-detection techniques such as those described in [7, 11]. Early detection of these safety-critical objects and increased attention to software operations involving them might forestal] safety-related software errors involving them.

3. *Use formal specification techniques in addition to naiural-language software requirements specifications.*

Lack of precision and incomplete requirements led to many of the safety-related software errors seen here. Enough detail is needed to cover all circumstances that can be envisioned (component failures, timing constraint violations, expired data) as well as to document all environmental assumptions (e.g., how close to the sun an instrument will point) and assumptions about other parts of the system (maximum transfer rate, consequences of race conditions or cycle slippage). The capability to describe dynamic events, the timing of process interactions in distinct computers, decentralized supervisory functions, etc., should be considered in chooosing a formal method [2,4, 5, 15,20,23].

4. Promote informal communica tion among teams.

Many safety-related software errors resulted from one individual or team misunderstanding a requirement or not knowing a fact about the system that member(s) of another development team knew. The goal is to be able to modularize responsibility in a development project without modularizing communication about the system under development. The identification and tracking of safety hazards in a system, for example, is clearly Lest done across team boundaries.

5. As requirements evolve, communicate the changes to the development and test teams.

This is both more important (because there are more requirements changes during design and testing) and more difficult (because of the number and size of the teams and the length of the development process) in a large, embedded system than in simpler systems. in analyzing the safety-r-elated software errors, it is evident that the determination as to who needs to know about a change is often made incorrectly. Frequently, changes that appear to involve only one team or system component end up affecting other teams or components at some later elate (sometimes as the result of incompatible changes in distinct units).

There is also a need for faster distribution} of changes that have been made, with the update stored so as to be fingertip accessible. CASE tools offer a possible solution to the difficulty of promulgating change without increasing paperwork.

The prevalence of safety-related software errors involving misunderstood or missing requirements points up the inadequacy of consistency checks of requirements and code as a means of demonstrating system correctness [1 O]. Code that implements incorrect requirements is incorrect if it fails to provide needed system behavior.

Similarly, generating test cases from misunderstood or missing requirements will not test system correctness. Traceability of requirements and automatic test generation from specifications offers only partial validation) of complex, embedded systems. Alternative validation and testing methods such as those described in [9, 11] offer greater coverage.

G. Include requirements for "defensive design" [1 7].

Many of the safety-related software errors involve inadequate software responses to extreme conditions or extreme values. Anomalous hardware behavior, unanticipated states, events out of order, and obsolete data arc all causes of safety-related software errors on the spacecraft.

Run-time safety checks on the validity of input data, watchdog timers, delay timers, software filters, software-imposed initialization conditions, additional exception handling, and assertion checking can be used to combat the many safety-critical software errors involving conditional and omission faults [1 1]. Requirements for error-llaIldliIlg, overflow protection, signal saturation limits, heartbeat and pulse frequency, maximum event duration, and system behavior under unexpected conditions can be added and traced into the design. Many sa.fety-related functional faults involve error-recovery routines being invoked inappropriately because of erroneous limit values or bad data.

Backward analysis from critical failures to possible causes offers one check of how defensive the requirements and design arc [1 2]. Requirements specifications that account for womt-case scenarios, models that can predict the range of possible (rather than allowable) values, and simulations that can discover unexpected interactions before system testing contribute to the system's defense against hazards.

B. Summary and Future Work

In large, embedded systems such as the two spacecraft in this study, the software requirements change throughout the software development process, even during system testing, This is largely due to unanticipated behavior, dynamic changes in the operating environmen t, and complex software/llardwarc and software/softwar-c interactions in the systems being developed. Controlling requirement changes (and, hence, the scope and cost of development) is difficult since the changes arc often prompted by an improved understanding of the software's necessary interfaces with the physical components of the spacecraft in which it is embedded. Complex timing issues and hardware idiosyncrasies often prompt changes to requirements or to design solutions.

The analysis presented here of the cause/effect relationships of safety-rclatccl software errors pinpoints aspects of system complexity which merit additional attention. Specifically, the results have shown that conditional faults (e.g., condition or limit values) are highly correlated with safety-related software errors. Operating faults (especially the omission of run-time reasonableness checks on data) arc also highly correlated with safety-rclatccl software errors. Unknown, undocumented, or erroneous requirements frequent] y arc associated with safety-related software errors as well. }lardwarc/software interfaces have been shown to be a frequent trouble spot because of the lack of communication between teams.

The results presented in this paper indicate a need for better methods to confront the real-world issues of developing safety-critical, embedded software in a complex, distributed system. Future work will be directed at incorporating knowledge of the distinct error mech - anisms that produce safety-related software errors into the requirements analysis and validation processes. Work is also needed on specifying how these results can be used to predict more precisely what features or combinations of factors in a safety-critical, embedded system arc likely to cause time-consulnirlg; and hazardous software errors.

References

[1] E. A. Addy, "A Case Study on Isolation of Safety-Critical Software," in *Proc 6th Annual Conf on Computer Assurance*. NIST/IEEE, 1991, pp. 75-83.

[2] A. M. Davis, *Software Requirements, Analysis and Specification*. Englewood Cliffs, N. J.: Prentice hall, 1990.

[3] D. E. Eckhardt, et al., "An Experimental Evaluation of Software Redundancy as a Strategy for improving Reliability," *IEEE Trans Software Eng*, 1 7, 7, July 199], pp. 692--702.

[4] A. Endres, "An Analysis of Errors ant] Their Causes in Systems Programs," *IEEE Trans Software Eng*, SE-1, 2, June 1975, pp. 140-149.

[5] E. M. Gray and R. H. Thayer, "Requirements," in *Aerospace Software Engineering, A Collection of Concepts.* Ed. C. Anderson and M. Dorfman. Washington: AIAA, 1991, pp. 89-121.

[6] ANSI/IEEE Standard Glossary of Software Engineering Terminology. New York: IEEE, 1983.

[7] M. S. Jaffe et al., "Software Requirements Analysis for Real-Time Process-Control Systems, " *IEEE Trans Software Eng, 17,* 3, March 1991, pp. 241-258.

[8] P. Jalote, *An Integrated Approach to Software Engineering. New* York: Springer-Verlag, 1991.

[9] J. C. Knight, "Testing," in *Aerospace Software Engineering, A Collection of Concepts.* Ed. C. Anderson and M. Dorfman. Washington: AIAA, 1991, pp. 135-159.

[10] N. G. Leveson, "Safety," in *Aerospace Software Engineering, A Collection of Concepts.* Ed. C. Anderson and M. Dorfman. Washington: AIAA, 1991, pp. 319-336.

[11] N. G. Leveson, "Software Safety in Embedded Computer Systems," *Commun ACM,* Vol. 34, No, 2, Feb 1991, pp. 35-46.

[12] N. G. Leveson and P. R. Harvey, "Analyzing Software Safety," *IEEE Transactions on Software Engineering*, SE-9, 5, Sept 1983, pp. 569-579.

[13] Karan L'Heureux, "Software Systems Safety Program RTOP, Phase A Report," Internal Document, Jet Propulsion laboratory, April 19, 1991.

[14] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proc IEEE Internat Symp on Requirements Engineering.* Los Alamitos, CA: IEEE Computer Society Press, 1993, pp. 126-133.

[15] R. Lutz and J. S. K. Wong, "Detecting Unsafe Error Recovery Schedules, " *IEEE Trans Software Eng*, 18, 8, Aug, 1992, pp. 749-760.

[16] T. Nakajo and H. Kume, "A Case History Analysis of Software Error Cause Effect Relationships," *IEEE Trans Software Eng* 17, 8, Aug 1991, pp. 830-838.

[17] P. G. Neumann, "The Computer-Related Risk of the Year: Weak Links and Correlated Events," in *Proc 6th Annual Conf on Computer Assurance.* NIST/IEEE, 1991, pp. 5-8.

[18] A. P. Nikora, "Error Discovery Rate by Severity Category and Time to Repair Software Failures for Three JPL Flight Projects," Internal Document, Jet Propulsion laboratory, 1991.

[19] T. J. Ostrand and E. J. Weyuker, "Collecting and Categorizing Software Error Data in an industrial Environment " *The Journal oj Systems and Software, 4, 1984, pp. 289-300.*

[20] *Proc Berkeley Workshop on Temporal and Real- Time Specification.* Eds. P. B. Ladkin and F. H. Vogt. Berkeley, CA: International Computer Science Institute, 1990, TR-90-060.

[21] N. F. Schneidewind and H.-M. Hoffmann, "An Experiment in Software Error Data Collection and Analysis," *IEEE Trans Software Eng*, SE-5, 3, May 1979, pp. 276-286.

[22] R. W. Selby and V. R. Basili, "Analyzing Error-Prone System Structure," *IEEE Trans Software Eng* 17, 2, Febr 1991, pp. 141-152.

[23] J. M. Wing, "A Specifier's introduction to Formal Methods," *Computer,* Vol. 23, Sept 1990, pp. 8-26.

Index Terms

Software errors, software safety, requirements analysis, embedded software, system testing, software specification, safety-critical systems, spacecraft.

Appendix

Table 1. Classification of Program Faults

	Program Faults				Safety-Related Program Faults			
Fault Types:	Voyager (134)		Galileo (253)		Voyager (74)		Galileo (121)	
Internal Faults	**1%**	**(1)**	3%	(7)	o%	(o)	2<%	(2)
Interface Faults	34%	**(46)**	18%	(47)	35%	(26)	19%	(23)
Functional Faults	**65%**	**(87)**	79%	(199)	65%	(48)	79%	(96)

“l-able 2. Classification of Functional Faults

	Functional Faults				Safety-Relateci Functional Faults			
Functional Fault Types:	Voyager (87)		Galileo (1 99)		Voyager (48)		Galileo (96)	
operating faults	22%	(19)	43%	(85)	19%	(9)	43%	(41)
Conditional Faults	26%	(23)	10%	(21)	**31%**	**(15)**	18%	**(17)**
Behavioral Faults	5296	(45)	47%	(93)	50%	(24)	40%	(38)

"1'able 3a. Relationships Causing Interface Faults

Root Causes (Human Errors):	Interface Faults: Voyager (46)		Interface Faults: Galileo (47)		Safety-Related Interface, Faults: Voyager (26)		Safety-Related Interface, Faults: Galileo (23)	
Communication Within Teams	7%	(3)	28%	(13)	8%	(2)	22%	(5)
Communication Between Teams: Misunderstood Hardware Interface Specifications	50%	(23)	42%	(20)	65%	(17)	48%	(11)
Misunderstood Software Interface Specifications	43%	(20)	30%	(14)	2.7%	(7)	30%	(7)

Table 3b. Relationships Causing Functional Faults

Root Causes (Human Errors):	Functional Faults: Voyager (87)		Functional Faults: Galileo **(199)**		Safety-Ralated Functional Faults: Voyager (48)		Safety-Ralated Functional Faults: Galileo (96)	
Requirement Recognition								
operating faults	9%	(8)	24%	(49)	8%	(4)	33%	(32)
Conditional faults	17%	(15)	8%	(16)	25%	(12)	16%	(15)
Behavioral faults	21%	(18)	30%	(59)	29%	(14)	30%	(29)
Total	47%	(41)	62%	(124)	62%	(30)	79%	(76)
Requirement Deployment								
Operating faults	13%	(11)	18%	(36)	11%	(5)	9%	(9)
Conditional faults	**9%**	**(8)**	3%	(5)	6%	(3)	2%	(2)
Behavioral faults	**31%**	**(27)**	17%	(34)	21%	(10)	9%	(9)
Total	**53%**	**(46)**	38%	(75)	38%	(18)	21%	(20)

Table 4a. Process Flaws Causing interface Faults

	Interface Faults				Safety-Related Interface Faults			
Process Flaws:	Voyager (46)		Galileo (47)		Voyager (26)		Galileo (23)	
1. Flaws in Control of System Complexity:								
1 -Interfaces not adequately identified or understood	70%	(32)	85%	(40)	54%	(14)	87%	(20)
2-Hardware behavior anomalies	30%	(14)	15'%	(7)	46%	(12)	13%	(3)
IIa. Flaws in Communication or Development Methods Causing Misunderstood Software Interfaces:								
1-Interface Specifications known but not documented or communicated	20%	(9)	38%	(18)	8%	(2)	35%	(8)
2-Interface design during testing	26%	(12)	2%	(1)	19%	(5)	0%	(0)
IIb. Flaws in Communication or Development Methods Causing Misunderstood Hardware Interfaces:								
1-Lack of communication between hardware and software teams	24%	(11)	28%	(13)	27%	(7)	3570	(8)
2-Hardware behavior not documented	30%	(14)	32%	(15)	46%	(12)	30%	(7)

Table 4b. Process Flaws Causing Functional Faults

	Functional Faults				Safety-Related Functional Faults			
Process Flaws:	Voyager (87)		Galileo (199)		Voyager (48)		Galileo *(96)*	
I. Flaws in Control of System Complexity:								
1-Requirements not identified: unknown, undocumented, or wrong	37%	(32)	53%	(105)	44%	(21)	60%	(58)
2-Requirements not understood	63%	(55)	47%	(94)	56%	(27)	40%	(38)
IIa. Flaws in Communication or Development Methods, causing errors in Recognizing Requirements:								
1-Specifications imprecise or unsystematic	16%	(14)	28%	(55)	21%	(10)	38%	(36)
2-Requirements missing from requirements documents	31%	(27)	35%	(69)	42%	(20)	42%	(40)
IIb. Flaws in Communication or Development Methods, causing errors in Deploying Requirements:								
1-Incomplete documentation of requirements or changes	6%	(5)	10%	(20)	2%	(1)	8%	(8)
2-Coding errors persisting until system testing	18%	(16)	9%	(18)	10%	(5)	**5%**	**(5)**
3-Design inadequate to perform required functions	29%	(25)	19%	(37)	25%	(12)	7%	(7)